Designing Modular Hardware Accelerators in C With ROCCC 2.0

Jason Villarreal, Adrian Park
Jacquard Computing Inc.
{jason, adrian}@jacquardcomputing.com

Walid Najjar, Robert Halstead
University of California, Riverside
{najjar, rhalstea}@cs.ucr.edu

***Abstract*—While FPGA-based hardware accelerators have repeatedly been demonstrated as a viable option, their programmability remains a major barrier to their wider acceptance by application code developers. These platforms are typically programmed in a low level hardware description language, a skill not common among application developers and a process that is often tedious and error-prone. Programming FPGAs from high level languages would provide easier integration with software systems as well as open up hardware accelerators to a wider spectrum of application developers.**

In this paper, we present a major revision to the Riverside Optimizing Compiler for Configurable Circuits (ROCCC) designed to create hardware accelerators from C programs. Novel additions to ROCCC include (1) intuitive modular bottom-up design of circuits from C, and (2) separation of code generation from specific FPGA platforms. The additions we make do not introduce any new syntax to the C code and maintain the high level optimizations from the ROCCC system that generate efficient code. The modular code we support functions identically as software or hardware. Additionally, we enable user control of hardware optimizations such as systolic array generation and temporal common subexpression elimination.

We evaluate the quality of the ROCCC 2.0 tool by comparing it to hand-written VHDL code. We show comparable clock frequencies and a 18% higher throughput. The productivity advantages of ROCCC 2.0 is evaluated using the metrics of lines of code and programming time showing an average of 15x improvement over hand-written VHDL.

***Keywords*-FPGAs, C-to-VHDL, Compilers, High Level Synthesis**

I. Introduction

Field programmable gate array (FPGA) based hardware accelerators are a viable option available to designers looking to improve performance of large software systems. The high performance computing domain contains many applications that take days or weeks to run and can benefit from a hardware implementation, including molecular dynamics simulations, genetic string matching, and XML query matching. While a custom hardware implementation will provide the most benefit, this approach is very expensive and time-intensive and not cost effective for most designers.

A wide variety of platforms incorporating both microprocessors and FPGAs are now available. The main difficulty of using FPGAs as hardware accelerators is in the programming of the configurable hardware. Hardware is commonly programmed with low level hardware description languages such as VHDL and Verilog. These languages require low-level knowledge such as complete timing information and can be tedious and error-prone to program in. Software application designers are typically unfamiliar with the requirements of a hardware design and have difficulty utilizing the hardware optimally.

As an alternative to hardware description languages, a wide assortment of high level languages have been developed to target FPGA-accelerators. Most of these high level languages (HLLs) are variants of C, as C is the most common language used in coding these applications. HLLs that generate hardware enable easier integration with existing software systems as well as open up hardware utilization to software designers who have no hardware design experience.

The standard approach taken in languages such as Impulse-C [21] and Handel-C [2] is to add explicit constructs that specify parallelism and timing to the C language in order to create hardware. An alternative, taken in toolsets such as Catapult-C [4] and Dime-C [14], is to target specific platforms and optimize strict subsets of C to take advantage of specific hardware.

In this paper, we introduce a compiler toolset, ROCCC 2.0, that takes a subset of C without the addition of explicit parallelism constructs and produces efficient hardware accelerators that are competitive with handwritten code and are not restricted to a particular platform. ROCCC 2.0 introduces a new way to generate circuits from high level languages that supports a modular bottom-up construction while maintaining top-down optimizations. This approach allows for an integrated reusability of compiled code as well as interchangeable software and hardware.

ROCCC 2.0 is an open source compiler built upon the SUIF [7] and LLVM [8] infrastructures and is available from http://roccc.cs.ucr.edu. The toolset is a continuation of the Riverside Optimizing Compiler for Configurable Circuits (ROCCC) [18] project.

The rest of this paper is organized as follows: Section 2 describes the original state of the ROCCC compiler as well as the problems that are alleviated by ROCCC 2.0. Section 3 describes the ROCCC 2.0 compiler additions, the interface abstractions that decouples the code generation from the specifics of a particular FPGA platform and the bottom-up creation process supported by the ROCCC 2.0 compiler and gives examples of using the ROCCC 2.0 compiler to create modular hardware accelerators. Section 4 compares several

examples with equivalent hand-written VHDL code. Section 5 discusses related work and we conclude in Section 6.

II. The ROCCC 1.0 Compiler

The Riverside Optimizing Compiler for Configurable Circuits (ROCCC) [18] is an open source compiler that accepts a strict subset of C and generates VHDL for use on FPGAs as hardware application accelerators. ROCCC was not designed to create hardware for entire applications, but instead focuses on the critical regions of large software systems. The critical regions typically consist of a loop nest performing extensive computation on large amounts of data.

ROCCC 1.0's design goals were to maximize throughput, minimize memory accesses, and minimize the size of the generated circuit. ROCCC does this by compiling code that leverages the strengths of FPGA hardware while restricting the type of code that is inefficient on FPGAs. Namely, ROCCC code takes advantage of the extensive amount of parallelism available on FPGAs and the ability to implement large computational pipelines on streams of data while attempting to minimize off-chip memory fetches and control flow, which are better handled on microprocessors.

The circuit generated by ROCCC is a decoupled architecture where the memory accesses have been separated from the data path. The data path consists of many unrolled loop bodies performing as much computation in parallel as possible each clock cycle given the area and bandwidth limitations of a specific platform. Data to feed the pipeline is fetched independently by a memory controller and pushed onto the data path when enough data is ready. After a set number of clock cycles, the data flows into output memory. The data path generated contains no control flow other than predication and the output is always generated after N clock cycles where N is the pipeline depth.

In generating the VHDL, the ROCCC compiler uses an intermediate format called CIRRF (Compiler Intermediate Representation for Reconfigurable Fabrics)[17]. CIRRF has two distinct representations, referred to as Hi-CIRRF and Lo-CIRRF. Hi-CIRRF is created after high level optimizations such as loop unrolling, constant propagation, and loop tiling are performed and consists mainly of C code with extra macros for identified hardware constructs. Lo-CIRRF is a low level representation of the code as a data flow graph with assembly like instructions and hardware components as the nodes in the graph. The Lo-CIRRF is mainly an internal representation that corresponds closely to the VHDL output by ROCCC.

The toolsets used in the ROCCC 1.0 compiler are the SUIF compiler [7] and the Machine-SUIF extension [3] to handle the Lo-CIRRF transformations. Both SUIF and Machine-SUIF are no longer supported and require older versions of common libraries in order to function correctly.

As shown in [23], generating hardware accelerators with ROCCC 1.0 for real software systems revealed several issues. In many cases, the ideal hardware algorithm is significantly different from the optimal software algorithm. High performance computing applications are tuned to get the best performance out of a sequential processor and do not port well to hardware as written. Additionally, compiler optimizations for software sometimes undermine the advantages hardware can utilize [22].

In designing hardware accelerators, it is advantageous to start out with an optimal hardware design in mind and have the software compile into that design. In order to do this, it was found that both applications and the compiler must be completely rewritten in order to generate the optimal design for hardware. A strict translation of the software system into hardware does not take advantage of the streaming nature of FPGAs and although functionally correct does not provide the speedup that was desired. Software applications are written with the assumption of a large memory hierarchy and must be algorithmically modified to support streaming. Creating a hardware accelerator that provided a speedup on an application required a complete tuning of both the application and the compiler to perform the proper optimizations to transform the algorithm into the appropriate hardware on a specific platform.

Additionally, each optimization in ROCCC 1.0 took the target platform into account. As different platforms were supported, the code base and complexity expanded exponentially. Unlike software compilation, taking the platforms into account resulted in completely different optimization flows. Compiling for one system might require a different order of optimizations and a different scope to each optimization. The complexity built into the compiler to support many platforms quickly became unmanageable, even if the underlying optimizations were the same.

While taking the platform into account can often produce better code, applying a standard set of optimizations targeting hardware and allowing the user to fine tune the optimizations for a specific platform can provide equivalent hardware. Additionally, user knowledge should be used to guide compilation when tradeoffs such as space versus throughput have to be decided and should not be decided by the compiler.

III. The ROCCC 2.0 Compiler

In order to address the issues faced by the original version of ROCCC, we have developed the ROCCC 2.0 compiler. Using the ROCCC compiler as a base, we have addressed the problem that specific hardware could not be generated without a complete overhaul of both application and compiler while maintaining the strengths, namely the extensive optimizations that produced efficient parallelized pipelines from C. We do this by supporting a new model in the ROCCC 2.0 compiler based upon *modularity* and *reusability*.

Since the SUIF platform is no longer supported, a major task in the creation of the ROCCC 2.0 compiler was rewriting ROCCC to work with a more modern compiler framework. We chose to implement the ROCCC 2.0 compiler using the Low Level Virtual Machine (LLVM) framework [8], which is currently being maintained and supported as well as being used by several major companies.

ROCCC 2.0 keeps the same design goals of maximizing throughput, minimizing memory accesses, and minimizing

the size of the generated circuit. As such, even though the generated code undergoes optimizations to reduce the size, no resource sharing is performed. The generated code prefers duplicated resources to increase performance. In the ideal case, the memory bandwidth of the circuit is saturated and the generated code will produce an entire loop iteration's worth of data every clock cycle, utilizing every generated component every clock cycle.

The ROCCC 2.0 compiler generates VHDL for two types of C code which we refer to as *modules* and *systems*.

One of the most critical features of a design environment is the ability to reuse sections of code. The ROCCC 2.0 compiler is built around the concept of reuse through the creation and integration of modules. Modules are concrete hardware blocks that have a known number of inputs and outputs, some internal computation, and a known delay. Modules written in C can be reused as both software functions or hardware functions and integrated directly into larger designs. Once compiled to VHDL, modules are available for integration in larger code through standard C function calls.

Information on all modules available to the compiler is stored in a standard database accessible via SQL queries. The database is implemented in SQLite3 [6] and allows users to see what cores are available in an external design environment. The database also allows designers to enter their own cores they might have access to, such as previously purchased IP cores or netlists. The designer needs to specify the interfaces, delay, and name of each externally added module. All the cores in the database are available to be integrated into pipelines created by the ROCCC 2.0 compiler.

System code refers to critical loops of applications that perform computations on large streams of data or windows in memory. System code can contain loops and reference memory through array accesses. Data reuse between consecutive iterations of the loop is detected and used to minimize memory fetches, with the necessary data elements being stored in a smart buffer [16]. If there is no data reuse, FIFOs are generated to fetch and store data. System code is very similar to the type of code supported by ROCCC 1.0 with the addition of modules.

Floating point operations and integer division are supported in the C code. The hardware support for these operations can come from a variety of known library components, as long as they are placed in the database of modules. In the experiments we ran, support for floating point operations and integer division came from cores generated by Xilinx Core Generator [10]. The specifics of delay and inputs/outputs were manually added to the database and exported to the ROCCC 2.0 compiler.

The ROCCC 2.0 compiler addresses the issue of platform tuning by removing the details of the platform from the compiler internals. Instead of generating code specific to a platform, the ROCCC 2.0 compiler creates circuits that interface to *streams* or *memories* through a *platform interface abstraction layer*. To compensate for the loss of the platform details, we give the user complete control over which optimizations are performed, what order, and the scope of each optimization. This control allows for the generation of circuits tuned to device specifics, such as unrolling to fill memory bandwidth and controlling the amount of computation per pipeline stage to control clock speed.

A. Platform Interface Abstraction Layer

Module code generated by the ROCCC 2.0 compiler is meant to be used either in other module code or in system code, and as such is simply a VHDL component with a variable number of inputs, outputs, and a set delay. System code is the only generated code that deals with reading and writing large amounts of external data. As such, system code needs to communicate with the world outside of the generated hardware. The ROCCC 2.0 compiler does not have any built in information regarding the specifics of the platform the generated code will run on. Instead, we generate hooks to communicate with the outside world as either a stream or as a memory. For every array used in the original C code, a hook is generated in the VHDL.

Streams are data busses that act as FIFOs, with the assumption that no data elements are read more than once. System code can handle reading N words per clock cycle from a single stream where N is a configurable amount the user specifies based upon the width of the actual stream on the platform being connected to.

Memory interfaces are generated when we must read a window of data that is noncontiguous, and an internal address generator is created as necessary. We treat memories as being word addressable, where the word size is determined by the size of the elements of the array in the original C. Again, ROCCC 2.0 can handle reading a configurable amount of words per clock cycle but can stall as necessary until data is available.

The number of outgoing memory requests is again configurable by the user. The addresses are generated independently of the data we receive, although we do require that the data is received in the order requested. This allows data to stream in at the maximum allowed rate without the overhead of handshaking if the platform allows it.

With both memory interfaces and stream interfaces, internal data reuse is handled through the creation of a smart buffer [16]. The smart buffer stores data that is used across loop iterations, allowing for the minimal amount of data to be read between activations of the data path. For example, if the data path requires nine elements from a three-by-three window as in the Max Filter System example, the smart buffer would store six elements and fetch three for consecutive data path invocations instead of fetching nine elements for consecutive data path invocations. This exists to minimize the number of memory fetches.

The ROCCC 2.0 compiler also supports the creation of channels between hardware systems. At a higher level, in the development environment, system code can be marked as receiving or sending data over channels, which in turn are translated into buffered FIFOs. These FIFOs are mapped

over some communication scheme and allow multi-FPGA communication.

The actual porting of generated code onto a platform is left to designers. As a proof of correctness, we have created two interfaces to both a high end system (the SGI Altix 4700 with RASC blade) [5] and a low end system (the Xilinx Virtex 5 ML507 board) [9].

In both cases, some VHDL glue logic is required as well as C code to replace the critical region with a call to hardware in the original application. The VHDL glue logic maps the hooks generated by the abstraction layer to actual components on the platform, namely the stream interfaces on the SGI-RASC and the PLB bus on the ML507 board. The C code is responsible for identifying arrays as input and output to the hardware and transferring this memory to and from the hardware accelerator. The time and complexity of this glue logic is dependent on the system being targeted, and may be automated for each platform. The glue logic will consist mainly of connections between the ROCCC platform interface and actual physical components on the system and small FSMs to handle buffering and timing issues.

B. Modular Design in C

Modules are defined using an interface/implementation dichotomy. Much like VHDL's entity/architecture pairing, a module is described with a struct that lists all of the inputs and outputs to the module. The inputs are identified by adding the suffix "_in" to the name of the variable while outputs are similarly identified using the suffix "_out."

The implementation is done as a function that takes an instance of this struct and returns an instance of this struct. All reads from input elements of this struct translate into reads from input ports in the generated hardware component and all writes to output elements of the struct become writes to output ports in the generated hardware component. Modules must contain only minimal control flow. Limited *if* statements are supported but loop constructs are not supported in modules.

Figure 1 shows an example of a module in C for the ROCCC 2.0 compiler. The module shown is an FFT module that takes three complex numbers and computes four outputs. The example shows integers as the inputs but floats and other concrete types are supported as well. Each output needs to have one assignment. Variables can be declared in the function as shown but are not visible outside the generated VHDL. The function for the implementation must return the same *struct* that is passed in. Enforcing this structure enables the code to work in C as well as be translated into hardware. A call to this function in the form "*f = FFT(f);*" will perform the same task in software as the hardware module.

Every module compiled with the ROCCC 2.0 compiler is exported for reuse in other ROCCC 2.0 codes. The appropriate information is placed in a header file as well as the database of cores. The exporting of a module creates a function that can be called from C to reference the hardware block that does not take a struct but instead takes individual variables that correspond to the modules inputs and outputs. Other C

```
// Interface
typedef struct
{
  int realOne_in ;
  int imagOne_in ;
  int realTwo_in ;
  int imagTwo_in ;
  int realOmega_in ;
  int imagOmega_in ;

  int A0_out ;
  int A1_out ;
  int A2_out ;
  int A3_out ;
} FFT_t ;

// Implementation
FFT_t FFT(FFT_t f)
{
  int tmp1 ;
  int tmp2 ;

  tmp1 = f.realOmega_in * f.realTwo_in ;
  tmp2 = f.imagOmega_in * f.imagTwo_in ;

  f.A0_out = f.realOne_in + tmp1 - tmp2 ;
  // The other outputs computations go here...
  return f ;
}
```

Recognized as inputs to the module

Recognized as outputs to the module

Internal registers

Fig. 1. An FFT module example

```
#include "roccc-library.h"

typedef struct
{
  int input0_in ;
  // Other inputs

  int tmp0_out ;
  // Other outputs
} FFTOneStage_t ;

FFTOneStage_t FFTOneStage(FFTOneStage_t t)
{
  FFT(t.input0_in,
      t.omega0_in,
      t.input16_in,
      t.omega1_in,
      t.input17_in,
      t.input1_in,
      t.temp0_out,
      t.temp1_out,
      t.temp2_out,
      t.temp3_out) ;

  // Others ...

  return t ;
}
```

Instantiation of module

Fig. 2. Calling the FFT module

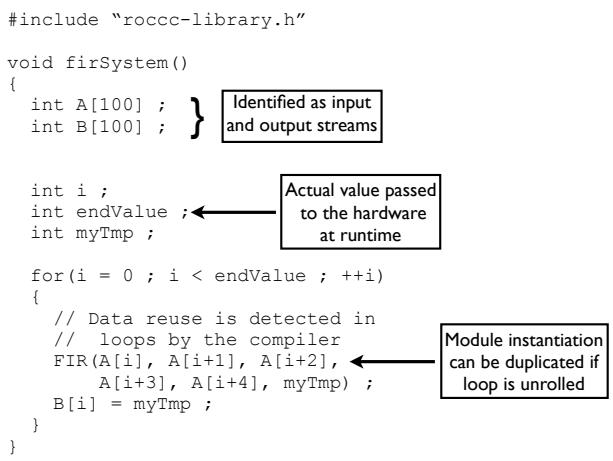

Fig. 3. System code for an FIR filter

code can make connections from their variables to the module inputs and outputs by calling this function.

Figure 2 shows an example module calling the FFT module from Figure 1. Invoking a module is done by calling the function corresponding to the module, located in the compiler maintained file *"roccc-library.h."* The inputs and outputs must be passed into the function in the order they were declared in the struct in the original module code. This struct is also listed in the *"roccc-library.h"* file. As shown in Figure 2, the inputs and outputs to the internal module may be mapped to the inputs and outputs of the containing module, although this is not necessary. In this example, the module is called with the inputs in this specific order to perform the FFT butterfly operation in hardware. When written in software, the butterfly operation requires shuffling large amounts of data between memory locations and passing arrays to functions. In hardware, these operations become wires, and those simple connections are reflected in the C code in Figure 2. The code shuffles the data through the values we associate with the internal modules and the butterfly operation is created.

System code must contain a loop of computation which may contain instantiations of modules. In system code, streams of data are represented by arrays and array accesses. Figure 3 shows an example system that performs the finite impulse response filter on a stream of data and generates an output stream. Modules inside of loops may be replicated if the compiler determines there are no data dependencies.

The complete design in both C code and generated hardware for the butterfly FFT is shown in Figure 4. The entire application accelerator is built from the bottom up as modules are written and included in larger modules and systems. At the bottom level, the FFT module from Figure 1 is compiled and exported by the ROCCC 2.0 compiler. Another module, FFTOneStage shown in Figure 2, uses multiple instances of the FFT module to perform one stage of the butterfly operation. At the topmost level, system code is written that instantiates several FFTOneStage modules and connects them in to an input stream and an output stream using arrays in the C code. The final structure of the generated hardware then consists of modules connected by wires in a criss-crossing pattern to perform the butterfly operation and has the input coming from a stream and the output flowing into a stream.

Note that, as mentioned above, a module can be imported as C or VHDL code or as a netlist. Importing a C module exposes its code to the ROCCC 2.0 compiler, by function inlining, and enables the optimizations to be carried out at the C code level. When a module is imported in VHDL it is treated as a black box by the ROCCC 2.0 compiler; however, its code is exposed to the synthesis tool. When a module is imported as a netlist its structure is preserved resulting in exact area and timing behavior.

C. Hardware Optimizations

Compiling C code for hardware enables the ROCCC compiler to perform several optimizations that produce better hardware but would not improve performance if the code was targeting a standard CPU. These hardware optimizations include systolic array generation and temporal common subexpression elimination.

1) Systolic Array Generation: Wavefront algorithms are typically coded in C as a nested for loop that iterates over a two-dimensional array. This construct is inefficient if converted directly into hardware and requires large memory allocations and transfers.

The ROCCC 1.0 compiler supported the generation of systolic arrays for wavefront algorithms as described in [12]. The set of transformations that enables the systolic array generation has been preserved and enhanced for ROCCC 2.0. Systolic array generation now may work with modules. Also, some steps that were previously left to the developer are now automated, including the transformation of a two-dimensional array with feedback into a streaming one dimensional array as input and a streaming one dimensional array as output.

2) Temporal Common Subexpression Elimination: Temporal Common Subexpression Elimination was introduced in [19] for the SA-C language. The optimization is much like common subexpression elimination, except the compiler looks across loop iterations to find computations that will be recomputed.

Support for temporal common subexpression elimination has been added to ROCCC 2.0. Commonalities across loop iterations are removed and replaced with feedback registers, reducing the amount of hardware generated. Additionally, module instantiations may be determined to be redundant across loop iterations and removed, greatly simplifying the generated hardware.

Temporal common subexpression elimination reduces the size of the hardware, but may change the clock rate resulting in less throughput, so this optimization is not performed unless the user specifies.

IV. Experimental Evaluation

We compiled several examples with the ROCCC 2.0 compiler and verified the generated circuits. These generated

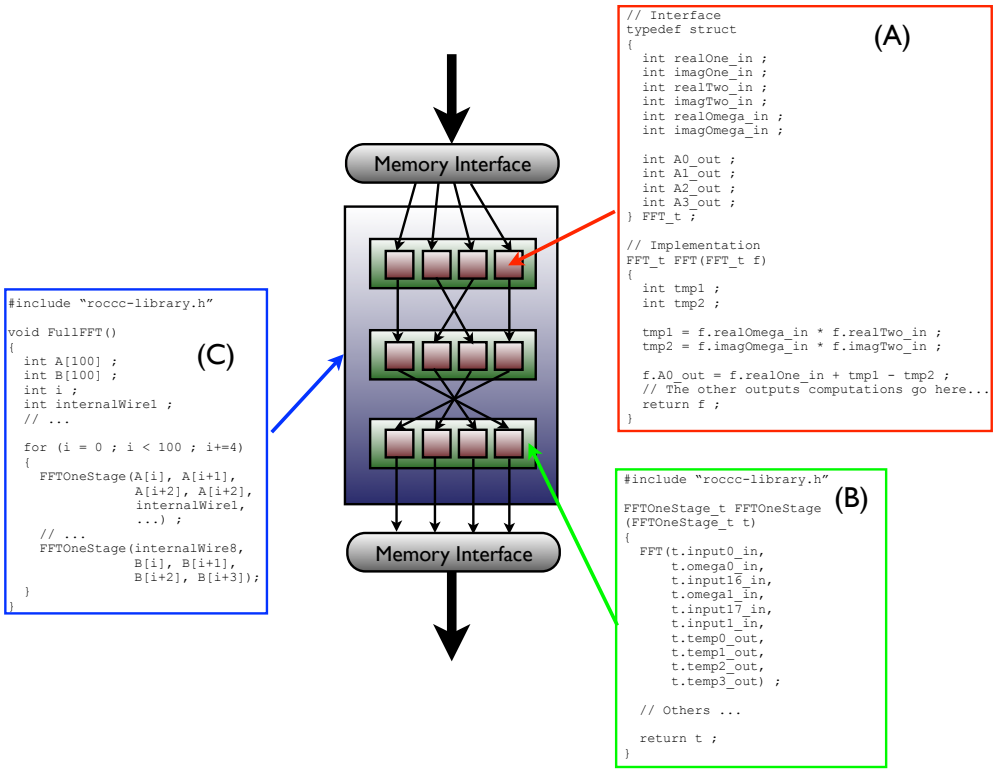

Fig. 4. Structure of generated system code of the butterfly FFT using modules. (A) is the FFT module that performs the base computation. (B) is the module for one stage in the butterfly that instantiates multiple FFT modules. (C) is the system code that instantiates multiple stages and connects the inputs and outputs to streams.

circuits were then synthesized with Synplify_Pro and placed and routed for an LX330 Virtex 5 FPGA.

The ten examples are a mixture of module and system code and reflect the additions we made to the compiler and the overall modular approach we are taking. The examples are:

- Complex multiplier: a module that performs the multiplication of two complex numbers with both an imaginary and a real component. The multiplication is done using three integer multiplications and four additions.
- FIR: a module that performs a single five-TAP filter on five numbers and generates a single output.
- FIR-System: a system that uses the FIR module to perform the filter on a window of five elements over a stream of data. The FIR-System example creates a stream interface and communicates with the outside world using a FIFO.
- Max Filter: a module that determines the maximum of three elements.
- Histogram: a module that starts out empty and picks up a valid value from a stream of data. The module then counts how many times that value is seen in the entire stream. When pipelined N times, the Histogram module can be used to determine the number of times N distinct variables appear in a stream of data.
- FFT: a module that computes the Fourier transform between two complex numbers and a complex omega. The module performs the base multiplications and additions of the FFT.
- FFTComplete: a module that pieces together 24 FFT modules into a complete butterfly circuit.
- MD: a module which performs the core calculations to determine the Coulombic force in three dimensions performed between two atoms at each time step in a molecular dynamics simulation.
- CompleteMD: a module which uses the MD module to calculate the Coulombic forces in three dimensions and also calculates the VanDerWaal energy between two atoms at each time step in a molecular dynamics simulation.
- SmithWaterman: system code which calculates a wavefront algorithm on a two dimensional array. This example works on 2 and 8-bit numbers and is transformed into a systolic array implementation by ROCCC 2.0 There are a total of 36 8-bit cells in the systolic array implementation after compilation.

The examples were written in C, compiled with ROCCC 2.0 and then synthesized into VHDL. The examples were also written directly in VHDL by hand, mimicking the functionality of the C code. While the handwritten VHDL did not necessarily follow the same structure as the generated VHDL, the functionality was identical and effort was made to produce high quality VHDL.

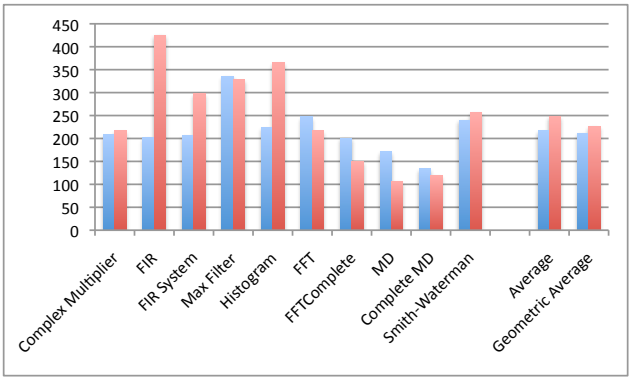

Fig. 5. Clock frequencies (MHz) of hand-written VHDL examples (blue) and ROCCC 2.0 generated examples (red)

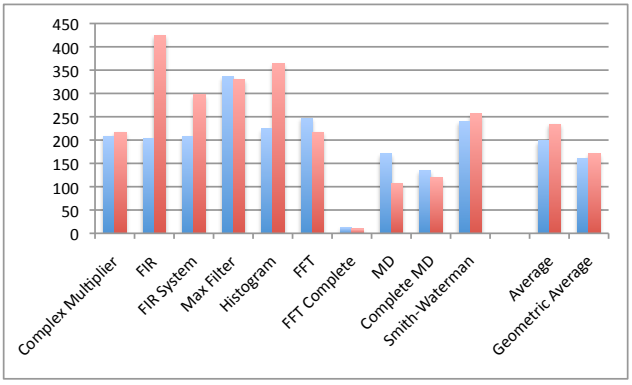

Fig. 6. Throughput (in millions of data elements per second) of ROCCC 2.0-generated and handwritten VHDL examples

Figure 5 shows the difference in the speed between the handwritten VHDL and the ROCCC 2.0 compiler generated VHDL in MHz. The circuits generated by the ROCCC 2.0 compiler are 6% faster than the corresponding handwritten VHDL on average. Taking the geometric mean to remove outliers, the speed of the generated circuit is on average 98% of the handwritten speed.

Figure 6 shows the maximum achievable throughput of each example in both the handwritten circuit and generated circuit.

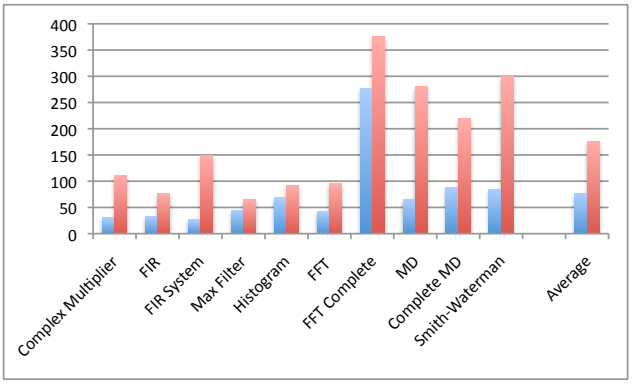

Fig. 7. Lines of C code (blue) and hand-written VHDL code (red).

This is calculated by assuming maximum memory saturation and determining how many clock cycles it takes before the circuit can generate data in the steady state (after the pipeline has been filled). The average throughput of the generated VHDL is 18% higher than the handwritten throughput. Using the geometric mean to eliminate outliers influence, we see that the throughput of the generated circuit is 6.98% higher than the handwritten circuits.

The sizes of the generated circuits created in terms of number of slices used is on average 1.53x more than the number of slices used in the handwritten VHDL. This size difference is due, in part, to the structure and optimizations done by ROCCC 2.0 to guarantee a higher throughput. The size of generated VHDL is expected to be an order of magnitude greater than handwritten code, and ours is much less while matching the throughput and clock speed of the handwritten code.

Figure 7 shows the difference in size of the examples, based off of lines of C and lines of VHDL. For each example, the amount of handwritten VHDL written is on average 4 times more than the amount of C.

The number of lines of generated VHDL is on average 2.65 times more than handwritten VHDL code. This bloat is partially due to assumptions made about the synthesis tool we made when designing the ROCCC 2.0 compiler. In some cases, extraneous VHDL is generated knowing that the synthesis tool will detect and optimize that code away to ease the process of code generation.

Furthermore, writing the small amount of C took on average under 5 minutes per example and the generated VHDL was functional without any modifications. Coding of the VHDL took on average 45 minutes per example and contained 1-2 errors that needed to be found before working correctly. These errors occurred during both simulation and post-simulation synthesis, causing additional time to re-synthesize. On average, each hand-written example took 75 minutes to produce a correct code. This is a factor of 15X in programming productivity.

In order to determine the effectiveness of the temporal common subexpression elimination (TCSE) optimization we compared the size of the generated circuit for a Max Filter System example before and after applying TCSE. The Max Filter System example uses four instances of the Max filter to determine the maximum value in a three-by-three 2D window.

Before applying TCSE, the amount of hardware generated was 851 slices. After applying TCSE, two modules were detected as redundant and removed resulting in 643 slices, or 75% of the original hardware while maintaining the same functionality.

V. Related Work

The C2R Compiler [13] creates Verilog from C code annotated with compiler directives that describe the desired architecture and parallelism. The process of creating hardware accelerators from C2R, covered in [11], requires adding explicit statements to handle concurrency and manual application

of loop unrolling to enable better pipelining. The ROCCC 2.0 compiler system requires no annotations to the C source in order to generate efficient pipelined hardware.

Impulse Accelerated Technologies produces the ImpulseC programming language [21], which is the commercialization of StreamsC [15]. Impulse concentrates on developing platform support packages to compile ImpulseC to different hardware platforms. ImpulseC requires defining C-level parallel processes and the communication channels between them. The details of the communication protocols specific to each platform are abstracted away using the Communicating Sequential Processes (CSP) model. ImpulseC requires minimal extensions to the C language in the form of new data types, including stream data types and predefined functions.

Handel-C, developed by Celoxica [2], has C-like syntax but very different semantics from C. Handel-C code is structured as sequential code with explicit parallel constructs such as channel communication and parallel sections. Keywords have been added the the C syntax and Handel-C code requires a cycle accurate description at the C level in order to generate hardware.

Catapult C, designed by Mentor Graphics [4], is a subset of C++ with no extensions. The code that can be compiled from Catapult C may be very general and may result in many different hardware implementations with vastly different timing and resource constraints. The Catapult C environment takes constraints and platform details in order to generate a set of Pareto-optimal implementations from which the user much choose. The ROCCC 2.0 compiler has removed the intricacy of the platform details and allows user control to generate the hardware they want.

The Dime-C language, created by Nallatech [14], is a subset of C that has several unique optimizations specific to certain platforms. The optimizations are enabled and supported by the underlying hardware platform also developed by Nallatech and not portable to other systems.

GAUT is high level synthesis tool dedicated to DSP applications [1]. The GAUT tool converts a C function into a pipelined architecture consisting of a processing unit, a memory unit, and a communication unit. The ROCCC 2.0 compiler creates as many memory interfaces as necessary while not specializing in DSP-based computation.

VI. Conclusions

In this paper we have introduced a new C to VHDL compilation tool that supports a novel way to create circuits from C. It allows the user to define self-contained modules that can be re-used by being imported into other modules or system codes. A module can be imported as C code, VHDL code or a netlist allowing pre-existing IP cores to be imported as modules. The creation and importing of hardware modules in C is done without the addition of explicit commands to C and allows software designers to generate hardware accelerators without the low level details inherent in hardware description languages. The ROCCC 2.0 compiler creates efficient pipelined circuits using these modules as well as integrating external IP into the same pipeline. We have shown that the circuits generated by the ROCCC 2.0 compiler are competitive with handwritten VHDL code in clock frequency and show at a productivity increase of 15x.

References

[1] GAUT - High Level Synthesis Tool from C to RTL. http://www-labsticc.univ-ubs.fr/www-gaut.

[2] Handel-C Overview. http://www.celoxica.com.

[3] Machine-SUIF. http://www.eecs.harvard.edu/hube/research/machsuif.html, 2004.

[4] Mentor Graphics. Catapult C Synthesis. http://www.mentor.com/c-design/, 2008.

[5] SGI Altix 4700. http://www.sgi.com/products/servers/altix/4700/, 2008.

[6] SQLite 3. http://www.sqlite.org.

[7] SUIF Compiler System. http://suif.stanford.edu/, 2004.

[8] The LLVM Compiler Infrastructure. http://llvm.org, 2008.

[9] Xilinx 5 ML507 Evaluation Board. http://www.xilinx.com/products/devkits/HW-V5-ML507-UNI-G.htm, 2009.

[10] Xilinx Core Generator. http://www.xilinx.com/ise/products/coregen_overview.pdf.

[11] S. Ahuja, S. Shukla, S. Gurumani, and C. Spackman. Hardware Coprocessor Synthesis from an ANSI C Specification. In *IEEE Design and Test Of Computers*, volume 26, 2009.

[12] A. Buyukkurt and W. Najjar. Compiler Generated Systolic Arrays For Wavefront Algorithm Acceleration on FPGAs. In *International Conference on Field Programmable Logic and Applications*, 2008.

[13] CebaTech. CebaTech C2R Compiler. http://www.cebatech.com/assets/files/C2R Compiler DataSheet final 022208f.pdf.

[14] G. Genest, R. Chamberlain, and R. Bruce. Programming an FPGA-Based Super Computer Using a C-to-VHDL Compiler: Dime-C. In *Second NASA/ESA Conference on Adaptive Hardware and Systems*, 2007.

[15] M. Gokhale, J. Stone, J. Arnold, and M. Lalinowski. Stream-oriented FPGA Computing in the Streams-C High Level Language. In *IEEE Symposium on FPGAs for Custom Computing Machines*, 2000.

[16] Z. Guo, A. Buyukkurt, and W. Najjar. Input Data Reuse in Compiling Window Operations Onto Reconfigurable Hardware. In *ACM Symposium on Languages, Compilers, and Tools for Embedded Systems*, 2004.

[17] Z. Guo, A. Buyukkut, J. Cortes, A. Mitra, and W. Najjar. A Compiler Intermediate Representation for Reconfigurable Fabrics. In *International Journal of Parallel Programming*, 2008.

[18] Z. Guo, W. Najjar, and A. Buyukkurt. Efficient Hardware Code Generation for FPGAs. In *ACM Transactions on Architecture and Compiler Optimizations*, 2008.

[19] J. Hammes, W. Bohm, C. Ross, M. Chawathe, B. Draper, R. Rinker, and W. Najjar. Loop Fusion and Temporal Common Subexpression Elimination in Window-based Loops. In *IPDPS 8th Reconfigurable Architectures Workshop*, 2001.

[20] A. Marongiu and P. Palazzari. The HARWEST Compiling Environment: Accessing the FPGA World through ANSI-C Programs. In *Proceedings of the Cray User Group*, 2008.

[21] D. Pellerein and S. Thibault. *Practical FPGA Programming in C*. Prentice Hall, 2005.

[22] G. Stitt and F. Vahid. New Decompilation Techniques for Binary-Level Co-Processor Generation. In *Proceedings of the 2005 IEEE/ACM International Conference on Computer-Aided Design*, 2005.

[23] J. Villarreal and W. Najjar. Exploration of Compiled Hardware Acceleration of Molecular Dynamics Code. In *Field Programmable Logic and Applications (FPL)*, 2008.